MA 3046 - Matrix Analysis
Laboratory Number 3
Practical Aspects of Numerical Linear Algebra

Practical Numerical Linear Algebra involves considerations different from, and generally well beyond, those which arise in introductory courses. The primary reason for these differing considerations is the fact that practical, "real world" applications involve matrices which tend to be very large, e.g. with 10^5 or more rows and columns. Such problems can only realistically be solved on computers. Unfortunately, solving problems on computers introduces a new set of concerns. Some of these simply relate to computational complexity, i.e. the number of computations required to implement algorithms on large matrices. As an example, we shall later show that the standard Gaussian elimination algorithm of introductory linear algebra for solving the square system of equations

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}$$

requires performing a number of multiplications, divisions, additions and subtractions (so-called floating point operations or flops) roughly equal to $\frac{2}{3}n^3$,where n is the number of rows and columns in $\mathbf{A}$. For a system where $n \sim 10^5$, this means approximately 667 *teraflops*, where one teraflop is a million megaflops, a rather daunting number.

However, flop counts are not the only issue in practical computation. As noted in class, execution times for linear algebra algorithms on modern computing platforms depend strongly not only on the flop count, but also on subtle interplays between numerous aspects of both software and hardware architecture, and on interrelationships of these to the size of the matrix. Software considerations include how efficiently the operating system manages cache; whether the programming language used is interpreted or compiled; whether the language uses row-oriented or column-oriented storage, and whether the language uses low-level, machine language coded primitive vector operations such as the Basic Linear Algebra System (BLAS) utilized by MATLAB. Hardware considerations focus on the speed of the CPU chip relative to the number of computations that must be performed; the CPU and system architecture in terms of either parallel or pipeline processing; the size and architecture of the CPU's arithmetic registers; the size of the CPU's on-chip cache; the size of the system RAM; and the effective speed of the system communication bus; and, last but not least, the effect on accuracy of the floating point number system.

To reiterate, in numerical linear algebra, we commonly use the term computational complexity as a synonym for a count of the number of flops required by a given calculation or algorithm. Moreover, as shown in class, hand-counting the number of flops associated with a given algorithm generally involves fairly straightforward, albeit sometimes laborious analysis. Furthermore, we have already seen, in a earlier lab, a very rudimentary use of MATLAB's relatively coarse **tic** and **toc** timing functions to analyze the growth in execution times related to the growth in computational complexity of the basic matrix multiplication, i.e.

$$\mathbf{b} * \mathbf{a}$$

```
MIL = 1500000;
NITER = 5 ;
data = [];
a = randn(MIL,1); b = randn(MIL,1);
c = zeros(size(a));
%
for n=1:10,
  tic
  for j=1:NITER; c = a+b ; end
  data = [ data; n, toc/NITER ];
end
%
for n= 1:10;
  tic
  for j=1:NITER; c = pi*a ; end
  data(n,3) = toc/NITER;
end
%
for n=1:10,
  tic
  for j=1:NITER; s = a'*b ; end
  data(n,4) = toc/NITER;
end
data
```

Figure 3.1 - Partial Listing of Program **time_prim.m**

In this lab we start with another very simple timing code, program **time_prim.m**, which is shown in Figure 3.1. This code computes the average time required for five iterations of each of the following operations

$$\mathbf{a} + \mathbf{b} \quad , \quad \pi\mathbf{a} \quad , \quad \text{and} \quad \mathbf{a}^H \mathbf{b}$$

where $\mathbf{a}$ and $\mathbf{b}$ have 1.5 *million* elements each. (Observe that basic computational complexity analysis indicates that, for general matrices of size n, these computations should require approximately n, n and $2n$ flops, respectively.) The results of this prove interesting!

As discussed in class, several aspects of the MATLAB (software) language can be expected to impact program execution in at least some situations. These aspects include

```
clear
%
NMAX = 500;
a = rand(NMAX);
b = a ;
%
timefull = [ ];              % initialize the data array
for ntest = 1:5             % perform five iterations
   tic                        % reset the "stopwatch"
   for i=1:NMAX               % perform the calculations
      for j=1:NMAX
         2*a(i,j);
      end
   end
   timefull = [ timefull , toc ]   % record the individual time
end
%
timerow = [ ];
for ntest = 1:5
   tic
   for i=1:NMAX
      2*a(i,:);
   end
   timerow = [ timerow , toc ]
end
```

Figure 3.2 - Partial Listing of Program **timing1.m**

the facts that MATLAB is an interpreted, column-oriented language, which also utilizes machine-language, matrix-primitive routines (the BLAS). The program **timing1.m**, part of which is shown in Figure 3.2, attempts to shed some light on how real these effects may be. Observe that the first portion of the code determines and records in an array the time required during each of five different iterations which use a "classic," FORTRAN-like double loop to calculate:

$$2\mathbf{A}$$

for a 500×500 matrix $\mathbf{A}$ composed of "random" numbers. As before, we use the initial call to **tic** in each iteration to effectively reset the "stopwatch" and then to **toc** at the end to record the elapsed time required by the double loop for that iteration. Similar logic in the second portion of the code repeats exactly the same calculation, but now on MATLAB indexed multiplication implemented on full row vectors (and therefore requiring only a single loop). Comparing the different times required by these two different "algorithms" should give us insights into the "cost" of explicit loops in MATLAB. Furthermore, it would be extremely simple (and is actually done in the full **timing1.m** code) to extend this test to both a single, column-oriented loop, in order to observe the differences, if any, between row and column-oriented performance. Finally, if we extend the code to time the single "pure," native MATLAB command **2∗a**, we should obtain some insight into the efficiency of native MATLAB, and perhaps observe some of the effects of the BLAS. (Of course, while running this code we should probably *simultaneously* observe the Windows Task Manager Performance Window Tab to ensure that differences in results are not due to other factors, e.g. memory-related problems.) Moreover, this simple example suggests any number of other tests. For example, we could observe how, if at all, the results of **timing1.m** change if, instead of simply multiplying $\mathbf{A}$ by two, we also stored the result in a new array, i.e. if we were to compute

$$\mathbf{B} = 2\mathbf{A}$$

(The difference here is that now MATLAB would also have to allocate storage for both the new and old matrices. This increased storage requirement could cause memory-related problems to arise here that were not present, or not at least until larger sized matrices, in the earlier case.)

We next turn to some of the practical numerical linear algebra considerations which arise due to the effects of floating-point computer arithmetic. This additional focus should seem natural and necessary, because, loosely speaking, while computational complexity plus software and other hardware concerns address the primary factors in how long we have to wait for an answer, considering floating-point arithmetic is essential to understand whether the answer we get will in fact be worth the wait!

As discussed in class, virtually all actual computations are done in a floating-point number system with some specified number of significant digits carried. (Specifically, PC-based systems utilize the IEEE 754-1985 standard system.) As discussed in class, any real number x and its floating-point representation are related by:

$$fl(x) = x(1+\delta) \ ,$$

where

$$|\delta| \ \le \ eps = \epsilon_{\text{machine}} \ \equiv \ \max_{x \neq 0} \left| \frac{x - fl(x)}{x} \right| = \begin{cases} \ \ \beta^{1-n} & \text{(chopping machine)} \\ \frac{1}{2}\beta^{1-n} & \text{(rounding machine)} \end{cases}$$

(Here β represents the number base for the machine, and n the number of significant digits carried in the floating point representation.) The quantity *eps* is generally referred to as *machine precision*, and on software for IEEE machines is often represented by a system variable with that name. We generally expect that any computer computation can involve errors on the order of magnitude of machine precision for that system (but hopefully not significantly larger), and numerical analysts devote a fair amount of effort to carefully analyzing the effects of such errors.

In practice, we can usually *calculate* the precision of a given machine relatively straightforwardly, using any one of several essentially equivalent definitions of machine precision. For example, machine precision can be defined alternatively as the largest relative error that will be produced by a change of one in the least significant digit, or as the smallest number δ such that:

$$fl\ (\ 1.\ +\ \delta\ )\ \neq\ 1.$$

The MATLAB script **fparith01.m** (Figure 3.3), utilizes the latter of these alternative definitions to determine, experimentally, MATLAB's effective machine precision. (Note that this program actually exits the loop at the point when a newly computed value of **a** first produces

$$fl(1. + a)\ =\ fl(1)\ \ ,$$

i.e. the way the logic of this program flows, we don't know we've reached machine precision until we pass it!)

```
 a = 1.0 ;
%
 while  ( (1. + a) ~= 1)
    a     = a/2.  ;
 end
%
 deltamin = 2.0*a ;
%
 sprintf(' Machine Precision of MATLAB  is  %9.2e', deltamin )
```

Figure 3.3 - Listing of Program **fparith01.m**

(There is one caution about using programs such as **fparith01.m** to determine machine precision in other situations. Some machines have very "smart," high-precision floating-point *coprocessors*, and some software is able to "look ahead" in computations. One has to test such systems and software fairly delicately, in order to avoid being mislead into confusing the accuracy of the coprocessor with that of the floating-point architecture of the underlying system.)

Unfortunately, MATLAB's relatively high default machine precision seldom allows us to observe meaningful effects from floating-point arithmetic in problems we can easily visualize. Fortunately, however, we can utilize the previously briefly-introduced and

```
  data = [] ;
%
  for NDIGITS = 2: 20 ;
%
    a = 1.0 ;
%
    while ( chop( (1.+a), NDIGITS ) ~= chop( (1.+a/2.), NDIGITS) )
        a  = chop( a/2. , NDIGITS) ;
    end
%
    theoret = 0.5*10^(1-NDIGITS) ;
    data = [ data ; NDIGITS   a   theoret ] ;
  end
%
%   Note the use of (semi)logarithmic plots is usually preferable
%  for displaying error behavior.
%
  semilogy( data(:,1) , data(:,2) , '*',  ...
          data(:,1) , data(:,3)  ) ;
```

Figure 3.4 - Listing of Program **fparith02.m**

slightly-misnamed MATLAB **chop( )** function to simulate reasonably well the effects on calculations of lower precisions. Specifically, as we saw before, the MATLAB statement

chop(x , n)

will *round* the floating-point number **x** to n significant digits. While, unfortunately for our purposes, **chop( )** only addresses the input argument **x**, its repeated use can reasonably mimic lower-precision machines. For example, consider **fparith02.m** (Figure 3.4). This script repeats the calculation of machine precision done in the **fparith01.m**, with rounding of each intermediate step now used to simulate an NDIGITS significant digit machines, and the simulated result compared to the theoretical one. (Note that **fparith02.m** does slightly change the stopping test from **fparith02.m**. You should convince yourself that stopping the loop now at the point where

$$fl(\ 1.\ +\ a\ )\ =\ fl(\ 1.\ +\ a/2.\ )\quad .$$

should not theoretically change in the answer.)

Machine precision represents the theoretical "worst case" error we expect in the floating-point *representation* of any number in a given system. But, as we know, that's only the beginning of the numerical accuracy "story," because subsequent floating-point *computations* involving these numbers can introduce *additional* errors. Fortunately, for the operations of multiplication, division and addition of two numbers with the same sign, these additional errors are generally only on the order of magnitude of machine precision and therefore not too serious. Unfortunately, the same is not necessarily true for subtraction (or addition of two numbers with different signs). Subtraction of two nearly equal numbers will cause cancellation of the most significant digits, and result in an answer with no, or almost no accurate digits. We commonly refer to this latter situation, when it occurs, as *catastrophic cancellation.* Moreover, the same floating-point calculation, computed on two different machines, or even using *two different compilers on the same machine*, may result in different answers, although in computationally "well-designed" expressions, these answers should not differ by more than a couple of orders of magnitude above machine precision. (A similar phenomenon can occur when different but algebraically equivalent formulas are used for the same quantity.)

We now turn to our primary area of interest - the effect(s) of floating-point arithmetic on the computations if linear algebra. For example, consider Gaussian elimination. In its most basic form, this algorithm provides an almost deceptively simple algorithm for solving this problem by reducing the $m \times (n+1)$ augmented matrix to (augmented) echelon form, i.e. by performing

$$[\,\mathbf{A} \vdots \mathbf{b}\,] \;\rightarrow\; [\,\mathbf{U} \vdots \tilde{\mathbf{b}}\,]$$

according to the basic pattern:

For each row, $i = 1, 2, \ldots, (m-1)$, in sequence,

(i) Find the leading non-zero element on the current (pivot) row and on every row *below* it

(ii) Interchange the current row with the row below, if any, whose leading non-zero is furthest to the left of the leading non-zero on the current row

(iii) Eliminate on every row below the (now) current (pivot) row using the elementary row operation:

$$R_k \leftarrow R_k - l_{kj} R_i \quad , \quad k = (i+1), \ldots, m$$

where $l_{kj} = \tilde{a}_{kj} / \tilde{a}_{ij}$

(where the tilde indicates elements in the augmented matrix that may have been changed from their original values in the matrix **A**.) This procedure was implemented in the **ge_steps.m** function introduced earlier in these labs.

Floating-point arithmetic introduces several potential problems for simple variants of Gaussian elimination. These problems revolve around several fundamental aspects of Gaussian elimination:

- First, the elementary row operation

$$R_k \leftarrow R_k - l_{kj} R_i$$

involves (potentially inaccurate) subtractions. Therefore, there is at least some risk that elimination may compute elements (including the pivots) that become increasingly inaccurate.

- Secondly, Gaussian elimination makes critical decisions, e.g. row interchanges, based upon whether or not element are **exactly** zero. However, floating-point arithmetic virtually guarantees that a number that should be exactly zero when computed in infinite precision arithmetic will not calculate to zero, but be only "small."
- Finally, because the Gaussian elimination multipliers (l_{kj}) are given by $\tilde{a}_{kj}/\tilde{a}_{ij}$, computations of
$$R_k \leftarrow R_k - l_{kj} R_i$$
based on a "small" pivot ($\tilde{a}_{ij}$) result in a very large multiplier (l_{kj}), which may result in the effective loss of most or even all of the "information" in R_k.

In this laboratory, we shall investigate the effects of floating-point arithmetic on Gaussian elimination with the help of two programs which use **chop()** to simulate the solution process on a appropriate notional, low-precision machine. A partial listing of the first program, **ge_steps_chop.m**, which implements Gaussian elimination with row interchanges only to avoid zero pivots, in this simulated finite-precision machine, is shown in Figure 3.5. This program requires only a minor modification of the program **ge_steps.m** we introduced in a earlier lab, and indicates how, in general, simulating finite-precision arithmetic need not be unduly difficult. Note that since the intended use of this program is only to study the effect of floating-point calculations on "small" matrices, we have not attempted to be the least bit efficient, e.g. we happily use double looks rather than more primitive expressions. A partial listing of the second program, **bwd_solve_chop.m**, which simulates the back-substitution on the same notional, low-precision machine, once the augmented matrix has been reduced to echelon form, is shown in Figure 3.6.

We would note, however, that both **ge_steps_chop.m** and **bwd_solve_chop.m** do incorporate one not quite so minor previously unused feature of MATLAB, i.e. the use of **global** variables. (Look at the first executable statement in the function.) Global variables are similar to COMMON variables in FORTRAN, and offer an alternative way for programs to pass data back and forth besides through the use of function arguments. Global variables offer a very powerful, but also very dangerous feature, because any program, script, or function can read *or write to* (i.e. change the value of) any variable defined as global in that program. Therefore, they should generally be employed on a strict "need to know" basis, and should be given very distinctive names so as to minimize the chances of their being accidentally or inadvertently changed.

We would again emphasize that while **m**-files such as **gepp_steps_chop.m** provide valuable windows into the inner workings of various algorithms, they are not designed to replace "commercial grade" codes. So for the solution of real, practical systems of linear equations, MATLAB's backslash function remains our choice (although we still have some things to learn about it).

```
   function [ uwork ] = ge_steps( aaug )
%
  global  NDIGITS
  if ( ~ exist('NDIGITS') )+( isempty('NDIGITS') )
     error('NDIGITS either not defined or not made global')
  end
%
 [m,n] = size(aaug) ;
 uwork = chop( aaug , NDIGITS ) ;
%
 working_row = 1;
 working_col = 1;
%
 while (working_row < m)*(working_col < n)
   i = working_row ;
   j = working_col ;
   [ maxcol , imax ] = max( abs( uwork(i:m, j))) ;
%
   if ( maxcol  == 0 ) ; %%  check for free column %%%
        working_col = working_col + 1 ;
%
     else   % So skip the rest if a free column was found
%
        if ( uwork(i,j) == 0 ) ; %%%%   check for zero pivot ;
          imax           = imax + i - 1 ;
          uwork( [i,imax], : ) = uwork( [imax, i], : ) ;
        end
%
        for ii =  i+1:m
           lcoef          =  chop( uwork(ii,j)/uwork(i,j), NDIGITS )  ;
           uwork(ii,j)      =  0 ;
           uwork(ii,j+1:n) = chop( uwork(ii,j+1:n) - ...
                     chop(lcoef*uwork(i,j+1:n), NDIGITS), NDIGITS ) ;
        end
        working_row = working_row + 1;
        working_col = working_col + 1;
   end
 end
```

Figure 3.5 - Partial Listing of Program **ge_steps_chop.m**

```
function  [ xvec ] = bwd_solve_chop( U , z )
%
  global  NDIGITS
%
  [ rowsU , colsU ] = size(U) ;
  uwork  = chop( U , NDIGITS ) ;
  zvec  = chop( z , NDIGITS ) ;
  xvec = zeros( rowsU, 1) ;
  xvec(rowsU) = chop( zvec(rowsU)/uwork(rowsU,rowsU) ,...
                                    NDIGITS ) ;
  for i = (rowsU-1):-1:1
     sum_lhs  = chop( uwork(i,(i+1):rowsU).*...
                      xvec((i+1):rowsU)', NDIGITS ) ;
     adj_num  = chop( zvec(i)-chop( sum(sum_lhs) , NDIGITS ) , ...
                              NDIGITS) ;
     xvec(i)  = chop( adj_num/uwork(i,i) , NDIGITS) ;
  end
%
```

Figure 3.6 - Partial Listing of Program **bwd_solve_chop.m**

Name: ____________________________

MA 3046 - Matrix Analysis
Laboratory Number 3
Practical Aspects of Numerical Linear Algebra

1. Start MATLAB and copy to your local directory the lab files:

   time_prim.m , **timing1.m** , **fparith01.m** , **fparith02.m** ,
   gepp_steps_chop.m and **bwd_solve_chop.m**

(Note that if you are not working on an NPS system, you may also have to download the file **chop.m** as well.) Also start the Windows Task Manager, select the Performance window, and then minimize the manager. (Make sure a small green CPU busy indicator appears in the lower right-hand portion of your screen!)

2. Determine the CPU speed and memory (RAM) in your computer:

   CPU Speed – ________________________

   Memory (RAM) – ________________________

3. Using the MATLAB editor, open the file **time_prim.m** and study it until you are reasonably convinced what it does. Then run it, and record representative *average* times for each of:

Vector Addition	–	______________________
Scalar Multiplication	–	______________________
Vector Inner Product	–	______________________

Briefly describe why these times either agree with, or do not agree with what you expected to see, based on the relative computational complexities.

If these results do not agree with what you expected to see, based on the relative computational complexities, propose one or more possible, reasonable explanations for the discrepancies.

4. Using the MATLAB editor, change the size of the vectors to some other reasonable, but still large size. Then rerun the program, recording the (new) representative *average* times for each of

Vector Addition	–	____________________
Scalar Multiplication	–	____________________
Vector Inner Product	–	____________________

What, if any thing has changed?

How do these new numbers either help to confirm or not confirm the tentative explanations you proposed in part 3.

5. Again using your web browser, open the script **timing1.m**, and examine it. Observe that there are two "operations" which impact the time this script should take to execute:

(i) Retrieving each element of the matrix **a** into the CPU.

(ii) Multiplying each retrieved element by two.

Then run program **timing1** and observe and record below average representative times for each method. (Note how each of the data arrays is built element by element during the loops. Also keep and eye on the CPU utilization meter from the Task Manager!)

Double Loop	–	____________________
Singe Row-Oriented Loop	–	____________________
Single Column-Oriented Loop	–	____________________
MATLAB Primitive	–	____________________

To what degree do these values either they confirm or appear to contradict earlier statements in class and in this laboratory about how MATLAB "works."

6. Rerecord your average representative times for each method and the 500×500 matrix below. Then, using MATLAB's text editor, change the value of **NMAX** in script **timing1.m** to 1500, and then to 2500. In each case, then rerun the script and record the corresponding representative average times below. (Again, in each case, keep an eye on the CPU utilization!)

Size	**Double Loop**	**Single Row – Orient. Loop**	**Single Column – Orient. Loop**	**MATLAB Primitive**
500	_______	_______	_______	_______
1500	_______	_______	_______	_______
2500	_______	_______	_______	_______

Based on this additional data, to what degree, if any, do you need to modify your earlier statement(s) about how these values either confirm or appear to contradict earlier statements in class and in this laboratory about how MATLAB "works."

7. Open the full Task Monitor window. (You can just double-click on the CPU utilization meter in the lower right screen.) Then switch to the MATLAB command window. (The Task Monitor window will probably stay on top, but don't worry about that!) In the MATLAB command window, give the command(s)

clear ; a = rand(1500) ; inv(a)

Observe both the CPU and memory utilization graphs, as well as the disk access light on the front of your PC. Basically, what do you see?

When MATLAB finished finding the inverse (i.e. when the ">>" prompt appears, give the command(s)

clear ; a = rand(3000) ; inv(a)

Again observe both the CPU and memory utilization graphs, as well as the disk access light on the front of your PC. Basically, what do you see now? (Unless you have **a lot** of time, and patience, after a couple of minutes of watching you'll probably have to use the Task Manager to "kill" MATLAB here!)

How do these latest results, compared with the 1500×1500 case agree or not agree with earlier statements in class and in this laboratory about how hardware architecture can effect execution speed?

8. At this point you'll probably need to restart MATLAB, or even reboot your system. (Sorry!) Once you've done this, open a texteditor window and examine the MATLAB script **fparith01.m**. What do you expect the result will be?

Then, give the MATLAB command

fparith01

(Or run the program directly from the editor window!) Compare this result with the number(s) given in the question(s) above? Does this result confirm what you inferred earlier about MATLAB's machine precision and what was discussed in class?

9. Give the command **help chop** and study the response.

a. Run commands, i.e.

chop(3.141592 , 3)	Answer – ____________________
chop(3.141592 , 4)	Answer – ____________________
chop(3.141592 , 5)	Answer – ____________________
chop(31.41592 , 5)	Answer – ____________________

and finally

chop(314.1592 , 5)	Answer – ____________________

b. Are the results what you expected?

10. Using another texteditor window, examine the MATLAB script **fparith02.m** (which you copied to your disk earlier).

a. What is this script basically doing, and what quantities is it computing?

b. What is the effect of the

data = [data ; NDIGITS   a   theoret]

statement?

11. Run program **fparith02**. Is the output what you expected? If not, why?

12. Use the MATLAB help command to determine the use of the **xlabel**, **ylabel** and **title** commands. Experiment with these to appropriately label this graph. When you're satisfied, print a hard copy of the final result.

13. Give the MATLAB commands

```
clear
global NDIGITS
NDIGITS = 3
```

Then create the 4×4 matrix ($\mathbf{a}$)

$$\mathbf{a} = \begin{bmatrix} 2.03 & 1.00 & 9.32 & 5.25 \\ -2.01 & -0.98 & -10.5 & -3.31 \\ 6.04 & 7.47 & 4.19 & 6.72 \\ 2.72 & 4.45 & 8.46 & 8.38 \end{bmatrix}$$

and the column vector:

$$\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

use the MATLAB backslash function to solve $\mathbf{a}\,\mathbf{x} = \mathbf{b}$, and record the result

$$\mathbf{x}_{true} = \begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix}$$

14. Then run the MATLAB program **ge_steps_chop.m** on the augmented matrix [**a** | **b**]. Look carefully for the appearance of small pivots, and the resulting effects. (Also see if you can produce, offline, by hand, a couple of the intermediate results.)

Record the final echelon augmented matrix:

$$\mathbf{uwork} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

Based on this result, use the **bwd_solve_chop.m** command, along with the proper submatrices of **uwork** to compute the solution resulting from using Gaussian elimination, without partial pivoting,in a three-digit, decimal, rounding machine:

$$\mathbf{x}_{3-\mathrm{digit}} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

(Again, you should make sure you can basically reproduce these values by hand computations simulating a three-digit, rounding machine.)